Правила тестирования в Java

**20 мин**

**24K**

Перевод

[Автор оригинала: Philipp Hauer](https://phauer.com/2019/modern-best-practices-testing-java/)

Чтобы покрытие кода было достаточным, а создание нового функционала и рефакторинг старого проходили без страха что-то сломать, тесты должны быть поддерживаемыми и легко читаемыми. В этой статье я расскажу о множестве приёмов написания юнит- и интеграционных тестов на Java, собранных мной за несколько лет. Я буду опираться на современные технологии: JUnit5, AssertJ, Testcontainers, а также не обойду вниманием Kotlin. Некоторые советы покажутся вам очевидными, другие могут идти вразрез с тем, что вы читали в книгах о разработке ПО и тестировании.  
  
Общие положения

Given, When, Then (Дано, Когда, То)

Используйте префиксы “actual\*” и “expected\*”

Используйте заданные значения вместо случайных

Пишите краткие и конкретные тесты

Где можно, используйте вспомогательные функции

Вспомогательные функции на Kotlin можно реализовать так:

Не злоупотребляйте переменными

Не расширяйте существующие тесты, чтобы «добавить ещё одну маленькую штучку»

Проверяйте только то, что хотите протестировать

Самодостаточные тесты

Не прячьте релевантные параметры (во вспомогательных функциях)

Держите тестовые данные внутри самих тестов

Используйте композицию вместо наследования

Прямолинейные тесты — это хорошо. Сравнивайте результат с константами

Не переиспользуйте продакшн-код

Не копируйте бизнес-логику в тесты

Не пишите слишком много логики

Запускайте тесты в среде, максимально похожей на боевую

Тестируйте максимально полную связку компонентов (интеграционные тесты вместо юнит тестов)

Не используйте in-memory-базы данных для тестов

Java/JVM

Используйте -noverify -XX:TieredStopAtLevel=1

Используйте AssertJ

Избегайте использовать assertTrue() и assertFalse()

Используйте JUnit5

Используйте параметризованные тесты

Группируйте тесты (@Nested)

Читаемые названия тестов при помощи @DisplayName или обратных кавычек в Kotlin

Имитируйте внешние сервисы (моки)

Используйте Awaitility для тестирования асинхронного кода

Не надо резолвить DI-зависимости (Spring)

Ваш код должен быть тестируемым

Не используйте статический доступ. Никогда

Параметризуйте

Используйте внедрение в конструктор

Не используйте Instant.now() или new Date()

Разделяйте асинхронное выполнение и собственно логику

Общие положения

Given, When, Then (Дано, Когда, То)

Тест должен содержать три блока, разделённых пустыми строками. Каждый блок должен быть максимально коротким. Используйте локальные методы для компактности записи.  
  
Given / Дано (ввод): подготовка теста, например, создание данных и конфигурация моков.  
When / Когда (действие): вызов тестируемого метода  
Then / То (вывод): проверка корректности полученного значения

// Правильно

@Test

**public** **void** **findProduct**() {

insertIntoDatabase(**new** Product(100, "Smartphone"));

Product product = dao.findProduct(100);

assertThat(product.getName()).isEqualTo("Smartphone");

}

Используйте префиксы “actual\*” и “expected\*”

// Неправильно

ProductDTO product1 = requestProduct(1);

ProductDTO product2 = **new** ProductDTO("1", List.of(State.ACTIVE, State.REJECTED))

assertThat(product1).isEqualTo(product2);

Если вы собираетесь использовать переменные в проверке на совпадение значений, добавьте к этим переменным префиксы “actual” и “expected”. Так вы улучшите читаемость кода и проясните назначение переменных. Кроме того, так их сложнее перепутать при сравнении.

// Правильно

ProductDTO actualProduct = requestProduct(1);

ProductDTO expectedProduct = **new** ProductDTO("1", List.of(State.ACTIVE, State.REJECTED))

assertThat(actualProduct).isEqualTo(expectedProduct); // ясно и красиво

Используйте заданные значения вместо случайных

Избегайте подавать случайные значения на вход тестов. Это может привести к «морганию» тестов, что чертовски сложно отлаживать. Кроме того, увидев в сообщении об ошибке случайное значение, вы не сможете проследить его до того места, где ошибка возникла.

// Неправильно

Instant ts1 = Instant.now(); // 1557582788

Instant ts2 = ts1.plusSeconds(1); // 1557582789

**int** randomAmount = **new** Random().nextInt(500); // 232

UUID uuid = UUID.randomUUID(); // d5d1f61b-0a8b-42be-b05a-bd458bb563ad

Используйте для всего подряд разные заранее заданные значения. Так вы получите идеально воспроизводимые результаты тестов, а также быстро найдёте нужное место в коде по сообщению об ошибке.

// Правильно

Instant ts1 = Instant.ofEpochSecond(1550000001);

Instant ts2 = Instant.ofEpochSecond(1550000002);

**int** amount = 50;

UUID uuid = UUID.fromString("00000000-000-0000-0000-000000000001");

Вы можете записать это ещё короче, используя вспомогательные функции (см. ниже).

Пишите краткие и конкретные тесты

Где можно, используйте вспомогательные функции

Вычленяйте повторяющийся код в локальные функции и давайте им понятные имена. Так ваши тесты будут компактными и легко читаемыми с первого взгляда.

// Неправильно

@Test

**public** **void** **categoryQueryParameter**() **throws** Exception {

List<ProductEntity> products = List.of(

**new** ProductEntity().setId("1").setName("Envelope").setCategory("Office").setDescription("An Envelope").setStockAmount(1),

**new** ProductEntity().setId("2").setName("Pen").setCategory("Office").setDescription("A Pen").setStockAmount(1),

**new** ProductEntity().setId("3").setName("Notebook").setCategory("Hardware").setDescription("A Notebook").setStockAmount(2)

);

**for** (ProductEntity product : products) {

template.execute(createSqlInsertStatement(product));

}

String responseJson = client.perform(get("/products?category=Office"))

.andExpect(status().is(200))

.andReturn().getResponse().getContentAsString();

assertThat(toDTOs(responseJson))

.extracting(ProductDTO::getId)

.containsOnly("1", "2");

}

// Правильно

@Test

**public** **void** **categoryQueryParameter2**() **throws** Exception {

insertIntoDatabase(

createProductWithCategory("1", "Office"),

createProductWithCategory("2", "Office"),

createProductWithCategory("3", "Hardware")

);

String responseJson = requestProductsByCategory("Office");

assertThat(toDTOs(responseJson))

.extracting(ProductDTO::getId)

.containsOnly("1", "2");

}

* используйте вспомогательные функции для создания данных (объектов) (createProductWithCategory()) и сложных проверок. Передавайте во вспомогательные функции только те параметры, которые релевантны в этом тесте, для остальных используйте адекватные значения по умолчанию. В Kotlin для этого есть дефолтные значения параметров, а в Java можно использовать цепочки вызова методов и перегрузку для имитации дефолтных параметров
* список параметров переменной длины сделает ваш код ещё изящнее (ìnsertIntoDatabase())
* вспомогательные функции также можно использовать для создания простых значений. В Kotlin это сделано ещё лучше через функции-расширения

// Правильно (Java)

Instant ts = toInstant(1); // Instant.ofEpochSecond(1550000001)

UUID id = toUUID(1); // UUID.fromString("00000000-0000-0000-a000-000000000001")

// Правильно (Kotlin)

**val** ts = 1.toInstant()

**val** id = 1.toUUID()

Вспомогательные функции на Kotlin можно реализовать так:

**fun** Int.**toInstant**(): Instant = Instant.ofEpochSecond(**this**.toLong())

**fun** Int.**toUUID**(): UUID = UUID.fromString("00000000-0000-0000-a000-${this.toString().padStart(11, '0')}")

Не злоупотребляйте переменными

Условный рефлекс у программиста — вынести часто используемые значения в переменные.

// Неправильно

@Test

**public** **void** **variables**() **throws** Exception {

String relevantCategory = "Office";

String id1 = "4243";

String id2 = "1123";

String id3 = "9213";

String irrelevantCategory = "Hardware";

insertIntoDatabase(

createProductWithCategory(id1, relevantCategory),

createProductWithCategory(id2, relevantCategory),

createProductWithCategory(id3, irrelevantCategory)

);

String responseJson = requestProductsByCategory(relevantCategory);

assertThat(toDTOs(responseJson))

.extracting(ProductDTO::getId)

.containsOnly(id1, id2);

}

Увы, это очень перегружает код. Хуже того, увидев значение в сообщении об ошибке, — его будет невозможно проследить до места, где ошибка возникла.

*«KISS важнее DRY»*

// Правильно

@Test

**public** **void** **variables**() **throws** Exception {

insertIntoDatabase(

createProductWithCategory("4243", "Office"),

createProductWithCategory("1123", "Office"),

createProductWithCategory("9213", "Hardware")

);

String responseJson = requestProductsByCategory("Office");

assertThat(toDTOs(responseJson))

.extracting(ProductDTO::getId)

.containsOnly("4243", "1123");

}

Если вы стараетесь писать тесты максимально компактно (что я, в любом случае, горячо рекомендую), то переиспользуемые значения хорошо видны. Сам код становится более убористым и хорошо читаемым. И, наконец, сообщение об ошибке приведёт вас точно к той строке, где ошибка возникла.

Не расширяйте существующие тесты, чтобы «добавить ещё одну маленькую штучку»

// Неправильно

**public** **class** **ProductControllerTest** {

@Test

**public** **void** **happyPath**() {

// здесь много кода...

}

}

Всегда есть соблазн добавить частный случай к существующему тесту, проверяющему базовую функциональность. Но в результате тесты становятся больше и сложнее для понимания. Частные случаи, раскиданные по большой простыне кода, легко не заметить. Если тест сломался, вы не сразу поймёте, что именно послужило причиной.

// Правильно

**public** **class** **ProductControllerTest** {

@Test

**public** **void** **multipleProductsAreReturned**() {}

@Test

**public** **void** **allProductValuesAreReturned**() {}

@Test

**public** **void** **filterByCategory**() {}

@Test

**public** **void** **filterByDateCreated**() {}

}

Вместо этого напишите новый тест с наглядным названием, из которого сразу будет понятно, какого поведения он ожидает от тестируемого кода. Да, придётся набрать больше букв на клавиатуре (против этого, напомню, хорошо способствуют вспомогательные функции), но зато вы получите простой и понятный тест с предсказуемым результатом. Это, кстати, отличный способ документировать новый функционал.

Проверяйте только то, что хотите протестировать

Думайте о том функционале, который тестируется. Избегайте делать лишние проверки просто потому, что есть такая возможность. Более того, помните о том, что уже проверялось в ранее написанных тестах и не проверяйте это повторно. Тесты должны быть компактными и их ожидаемое поведение должно быть очевидным и лишённым ненужных подробностей.  
  
Предположим, что мы хотим проверить HTTP-ручку, возвращающую список товаров. Наш тестовый набор должен содержать следующие тесты:  
  
1. Один большой тест маппинга, который проверяет, что все значения из БД корректно возвращаются в JSON-ответе и правильно присваиваются в нужном формате. Мы легко можем написать это при помощи функций isEqualTo() (для единичного элемента) или containsOnly() (для множества элементов) из пакета AssertJ, если вы правильно реализуете метод equals().

String responseJson = requestProducts();

ProductDTO expectedDTO1 = **new** ProductDTO("1", "envelope", **new** Category("office"), List.of(States.ACTIVE, States.REJECTED));

ProductDTO expectedDTO2 = **new** ProductDTO("2", "envelope", **new** Category("smartphone"), List.of(States.ACTIVE));

assertThat(toDTOs(responseJson))

.containsOnly(expectedDTO1, expectedDTO2);

2. Несколько тестов, проверяющих корректное поведение параметра ?category. Здесь мы хотим проверить только правильную работу фильтров, а не значения свойств, потому что мы сделали это раньше. Следовательно, нам достаточно проверить совпадения полученных id товаров:

String responseJson = requestProductsByCategory("Office");

assertThat(toDTOs(responseJson))

.extracting(ProductDTO::getId)

.containsOnly("1", "2");

3. Ещё пару тестов, проверяющих особые случаи или особую бизнес-логику, например что определённые значения в ответе вычислены корректно. В этом случае нас интересуют только несколько полей из всего JSON-ответа. Тем самым мы своим тестом документируем именно эту специальную логику. Понятно, что ничего кроме этих полей нам здесь не нужно.

assertThat(actualProduct.getPrice()).isEqualTo(100);

Самодостаточные тесты

Не прячьте релевантные параметры (во вспомогательных функциях)

// Неправильно

insertIntoDatabase(createProduct());

List<ProductDTO> actualProducts = requestProductsByCategory();

assertThat(actualProducts).containsOnly(**new** ProductDTO("1", "Office"));

Использовать вспомогательные функции для генерации данных и проверки условий удобно, но их следует вызывать с параметрами. Принимайте на вход параметры для всего, что значимо в рамках теста и должно контролироваться из кода теста. Не заставляйте читателя переходить внутрь вспомогательной функции, чтобы понять смысл теста. Простое правило: смысл теста должен быть понятен при взгляде на сам тест.

// Правильно

insertIntoDatabase(createProduct("1", "Office"));

List<ProductDTO> actualProducts = requestProductsByCategory("Office");

assertThat(actualProducts).containsOnly(**new** ProductDTO("1", "Office"));

Держите тестовые данные внутри самих тестов

Всё должно быть внутри. Велик соблазн перенести часть данных в метод @Before и переиспользовать их оттуда. Но это вынудит читателя скакать туда-сюда по файлу, чтобы понять, что именно тут происходит. Опять же, вспомогательные функции помогут избежать повторений и сделают тесты более понятными.

Используйте композицию вместо наследования

Не выстраивайте сложных иерархий тестовых классов.

// Неправильно

**class** **SimpleBaseTest** {}

**class** **AdvancedBaseTest** **extends** **SimpleBaseTest** {}

**class** **AllInklusiveBaseTest** **extends** **AdvancedBaseTest** {}

**class** **MyTest** **extends** **AllInklusiveBaseTest** {}

Такие иерархии усложняют понимание и вы, скорее всего, быстро обнаружите себя пишущим очередного наследника базового теста, внутри которого зашито множество хлама, который текущему тесту вовсе не нужен. Это отвлекает читателя и приводит к трудноуловимым ошибкам. Наследование не гибко: как вы сами думаете, можно ли использовать все методы класса AllInclusiveBaseTest, но ни одного из его родительского AdvancedBaseTest? Более того, читателю придётся постоянно прыгать между различными базовыми классами, чтобы понять общую картину.

*«Лучше продублировать код, чем выбрать неправильную абстракцию»* (Sandi Metz)

Вместо этого я рекомендую использовать композицию. Напишите маленькие фрагменты кода и классы для каждой задачи, связанной с фикстурами (запустить тестовую базу данных, создать схему, вставить данные, запустить мок-сервер). Переиспользуйте эти запчасти в методе @BeforeAll или через присвоение созданных объектов полям тестового класса. Таким образом, вы сможете собирать каждый новый тестовый класс из этих заготовок, как из деталей Лего. В результате каждый тест будет иметь свой собственный понятный набор фикстур и гарантировать, что в нём не происходит ничего постороннего. Тест становится самодостаточным, потому что содержит в себе всё необходимое.

// Правильно

**public** **class** **MyTest** {

// композиция вместо наследования

**private** JdbcTemplate template;

**private** MockWebServer taxService;

@BeforeAll

**public** **void** **setupDatabaseSchemaAndMockWebServer**() **throws** IOException {

**this**.template = **new** DatabaseFixture().startDatabaseAndCreateSchema();

**this**.taxService = **new** MockWebServer();

taxService.start();

}

}

// В другом файле

**public** **class** **DatabaseFixture** {

**public** JdbcTemplate **startDatabaseAndCreateSchema**() **throws** IOException {

PostgreSQLContainer db = **new** PostgreSQLContainer("postgres:11.2-alpine");

db.start();

DataSource dataSource = DataSourceBuilder.create()

.driverClassName("org.postgresql.Driver")

.username(db.getUsername())

.password(db.getPassword())

.url(db.getJdbcUrl())

.build();

JdbcTemplate template = **new** JdbcTemplate(dataSource);

SchemaCreator.createSchema(template);

**return** template;

}

}

И ещё раз:

*«KISS важнее DRY»*

Прямолинейные тесты — это хорошо. Сравнивайте результат с константами

Не переиспользуйте продакшн-код

Тесты должны проверять продакшн-код, а не переиспользовать его. Если вы переиспользуете боевой код в тесте, вы можете пропустить баг в этом коде, потому что больше не тестируете его.

// Неправильно

**boolean** isActive = **true**;

**boolean** isRejected = **true**;

insertIntoDatabase(**new** Product(1, isActive, isRejected));

ProductDTO actualDTO = requestProduct(1);

// переиспользование боевого кода

List<State> expectedStates = ProductionCode.mapBooleansToEnumList(isActive, isRejected);

assertThat(actualDTO.states).isEqualTo(expectedStates);

Вместо этого при написании тестов думайте в терминах ввода и вывода. Тест подаёт данные на вход и сравнивает вывод с предопределёнными константами. Бóльшую часть времени переиспользование кода не требуется.

// Do

assertThat(actualDTO.states).isEqualTo(List.of(States.ACTIVE, States.REJECTED));

Не копируйте бизнес-логику в тесты

Маппинг объектов — яркий пример случая, когда тесты тащат в себя логику из боевого кода. Предположим наш тест содержит метод mapEntityToDto(), результат выполнения которого используется для проверки, что полученный DTO содержит те же значения, что и элементы, которые были добавлены в базу в начале теста. В этом случае вы, скорее всего, скопируете в тест боевой код, который может содержать ошибки.

// Неправильно

ProductEntity inputEntity = **new** ProductEntity(1, "envelope", "office", **false**, **true**, 200, 10.0);

insertIntoDatabase(input);

ProductDTO actualDTO = requestProduct(1);

// mapEntityToDto() содержит ту же логику, что и продакшн-код

ProductDTO expectedDTO = mapEntityToDto(inputEntity);

assertThat(actualDTO).isEqualTo(expectedDTO);

Правильным будет решение, при котором actualDTO сравнивается с созданным вручную эталонным объектом с заданными значениями. Это предельно просто, понятно и защищает от потенциальных ошибок.

// Правильно

ProductDTO expectedDTO = **new** ProductDTO("1", "envelope", **new** Category("office"), List.of(States.ACTIVE, States.REJECTED))

assertThat(actualDTO).isEqualTo(expectedDTO);

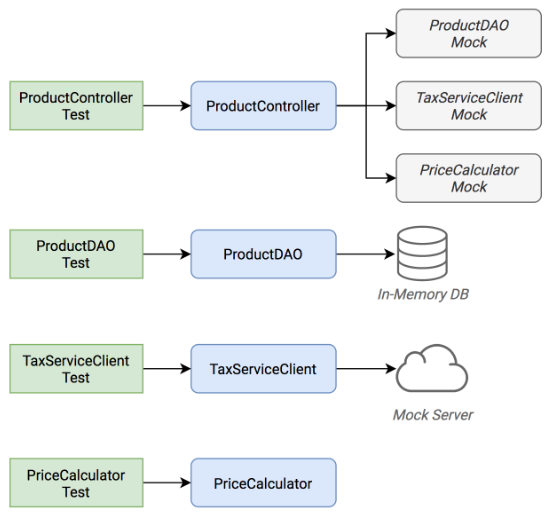
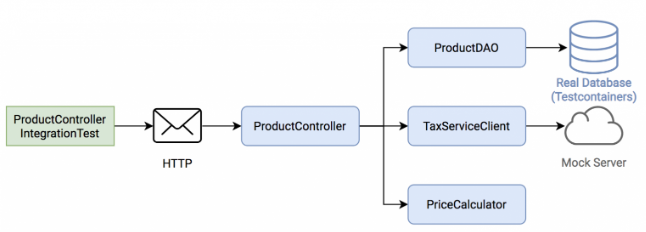
Если вы не хотите создавать и проверять на совпадение целый эталонный объект, можете проверить дочерний объект или вообще только релевантные тесту свойства объекта.

Не пишите слишком много логики

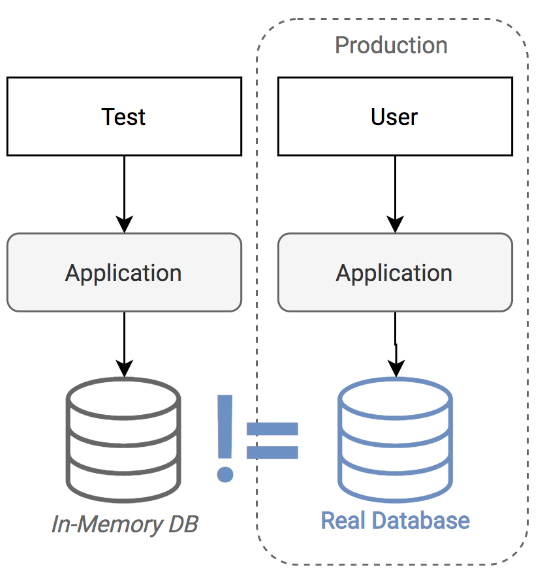
Напомню, что тестирование касается в основном ввода и вывода. Подавайте на вход данные и проверяйте, что вам вернулось. Нет необходимости писать сложную логику внутри тестов. Если вы вводите в тест циклы и условия, вы делаете его менее понятным и более неустойчивым к ошибкам. Если ваша логика проверки сложна, пользуйтесь многочисленными функциями AssertJ, которые сделают эту работу за вас.

Запускайте тесты в среде, максимально похожей на боевую

Тестируйте максимально полную связку компонентов

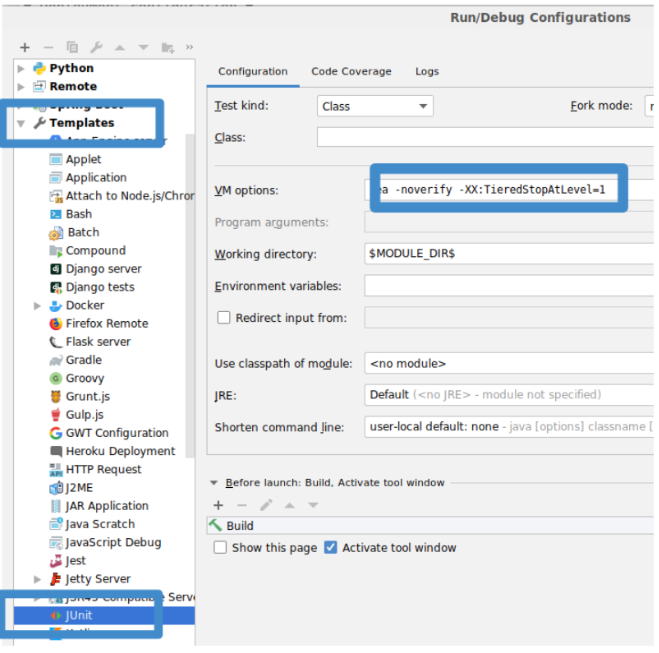
Обычно рекомендуется тестировать каждый класс изолированно при помощи моков. У этого подхода, однако, есть и недостатки: таким образом не тестируется взаимодействие классов между собой, и любой рефакторинг общих сущностей сломает все тесты разом, потому что у каждого внутреннего класса свои тесты. Кроме того, если писать тесты для каждого класса, то их будет просто слишком много.  
  
  
*Изолированное юнит-тестирование каждого класса*  
  
Вместо этого я рекомендую сосредоточиться на интеграционном тестировании. Под «интеграционным тестированием» я подразумеваю сбор всех классов воедино (как на продакшене) и тестирование всей связки, включая инфраструктурные компоненты (HTTP-сервер, базу данных, бизнес-логику). В этом случае вы тестируете поведение вместо реализации. Такие тесты более аккуратны, близки к реальному миру и устойчивы к рефакторингу внутренних компонентов. В идеале, вам будет достаточно одного класса тестов.  
  
   
*Интеграционное тестирование (= собрать все классы вместе и тестировать связку)*

Не используйте in-memory-базы данных для тестов

  
*С in-memory базой вы тестируете не в той среде, где будет работать ваш код*  
  
Используя in-memory базу ([H2](https://h2database.com/html/main.html), [HSQLDB](http://hsqldb.org/), [Fongo](https://github.com/fakemongo/fongo)) для тестов, вы жертвуете их достоверностью и рамками применимости. Такие базы данных часто ведут себя иначе и выдают отличающиеся результаты. Такой тест может пройти успешно, но не гарантирует корректной работы приложения на проде. Более того, вы можете запросто оказаться в ситуации, когда вы не можете использовать или протестировать какое-то характерное для вашей базы поведение или фичу, потому в in-memory БД они не реализованы или ведут себя иначе.  
  
Решение: использовать такую же БД, как и в реальной эксплуатации. Замечательная библиотека [Testcontainers](https://www.testcontainers.org/) предоставляет богатый API для Java-приложений, позволяющий управлять контейнерами прямо из кода тестов.

Java/JVM

Используйте -noverify -XX:TieredStopAtLevel=1

Всегда добавляйте опции JVM -noverify -XX:TieredStopAtLevel=1 в вашу конфигурацию для запуска тестов. Это сэкономит вам 1-2 секунды на старте виртуальной машины перед тем, как начнётся выполнение тестов. Это особенно полезно на начальной стадии работы над тестами, когда вы часто запускаете их из IDE.  
  
Обратите внимание, что начиная с Java 13 -noverify объявлен устаревшим.  
  
Совет: добавьте эти аргументы к шаблону конфигурации “JUnit” в IntelliJ IDEA, чтобы не делать это каждый раз при создании нового проекта.  
  


Используйте AssertJ

[AssertJ](http://joel-costigliola.github.io/assertj/) — исключительно мощная и зрелая библиотека, обладающая развитым и безопасным API, а также большим набором функций проверки значений и информативных сообщений об ошибках тестирования. Множество удобных функций проверки избавляет программиста от необходимости описывать комплексную логику в теле тестов, позволяя делать тесты лаконичными. Например:

assertThat(actualProduct)

.isEqualToIgnoringGivenFields(expectedProduct, "id");

assertThat(actualProductList).containsExactly(

createProductDTO("1", "Smartphone", 250.00),

createProductDTO("1", "Smartphone", 250.00)

);

assertThat(actualProductList)

.usingElementComparatorIgnoringFields("id")

.containsExactly(expectedProduct1, expectedProduct2);

assertThat(actualProductList)

.extracting(Product::getId)

.containsExactly("1", "2");

assertThat(actualProductList)

.anySatisfy(product -> assertThat(product.getDateCreated()).isBetween(instant1, instant2));

assertThat(actualProductList)

.filteredOn(product -> product.getCategory().equals("Smartphone"))

.allSatisfy(product -> assertThat(product.isLiked()).isTrue());

Избегайте использовать assertTrue() и assertFalse()

Использование простых assertTrue() или assertFalse() приводит к загадочным сообщениям об ошибках тестов:

// Неправильно

assertTrue(actualProductList.contains(expectedProduct));

assertTrue(actualProductList.size() == 5);

assertTrue(actualProduct **instanceof** Product);

expected: <**true**> but was: <**false**>

Используйте вместо них вызовы AssertJ, которые «из коробки» возвращают понятные и информативные сообщения.

// Правильно

assertThat(actualProductList).contains(expectedProduct);

assertThat(actualProductList).hasSize(5);

assertThat(actualProduct).isInstanceOf(Product.class);

Expecting:

<[Product[id=1, name='Samsung Galaxy']]>

to contain:

<[Product[id=2, name='iPhone']]>

but could not find:

<[Product[id=2, name='iPhone']]>

Если вам надо проверить boolean-значение, сделайте сообщение более информативным при помощи метода as() AssertJ.

Используйте JUnit5

[JUnit5](http://junit.org/junit5/) — превосходная библиотека для (юнит-)тестирования. Она находится в процессе постоянного развития и предоставляет программисту множество полезных возможностей, таких, например, как параметризованные тесты, группировки, условные тесты, контроль жизненного цикла.

Используйте параметризованные тесты

Параметризованные тесты позволяют запускать один и тот же тест с набором различных входных значений. Это позволяет проверять несколько кейсов без написания лишнего кода. В JUnit5 для этого есть отличные инструменты @ValueSource, @EnumSource, @CsvSource и @MethodSource.

// Правильно

@ParameterizedTest

@ValueSource(strings = ["§ed2d", "sdf\_", "123123", "§\_sdf\_\_dfww!"])

**public** **void** **rejectedInvalidTokens**(String invalidToken) {

client.perform(get("/products").param("token", invalidToken))

.andExpect(status().is(400))

}

@ParameterizedTest

@EnumSource(WorkflowState::class, mode = EnumSource.Mode.INCLUDE, names = ["FAILED", "SUCCEEDED"])

**public** **void** **dontProcessWorkflowInCaseOfAFinalState**(WorkflowState itemsInitialState) {

// ...

}

Я горячо рекомендую использовать этот приём по максимуму, поскольку он позволяет тестировать больше кейсов с минимальными трудозатратами.  
  
Наконец, я хочу обратить ваше внимание на @CsvSource и @MethodSource, которые можно использовать для более сложной параметризации, где также надо контролировать результат: вы можете передать его в одном из параметров.

@ParameterizedTest

@CsvSource({

"1, 1, 2",

"5, 3, 8",

"10, -20, -10"

})

**public** **void** **add**(**int** summand1, **int** summand2, **int** expectedSum) {

assertThat(calculator.add(summand1, summand2)).isEqualTo(expectedSum);

}

@MethodSource особенно эффективен в связке с отдельным тестовым объектом, содержащим все нужные параметры и ожидаемые результаты. К сожалению, в Java описание таких структур данных (т.н. POJO) очень громоздки. Поэтому я приведу пример с использованием дата-классов Kotlin.

data class **TestData**(

val input: String?,

val expected: Token?

)

@ParameterizedTest

@**MethodSource**("validTokenProvider")

fun `parse valid tokens`(data: TestData) {

assertThat(parse(data.input)).isEqualTo(data.expected)

}

**private** fun **validTokenProvider**() = Stream.of(

TestData(input = "1511443755\_2", expected = Token(1511443755, "2")),

TestData(input = "151175\_13521", expected = Token(151175, "13521")),

TestData(input = "151144375\_id", expected = Token(151144375, "id")),

TestData(input = "15114437599\_1", expected = Token(15114437599, "1")),

TestData(input = **null**, expected = **null**)

)

Группируйте тесты

Аннотация @Nested из JUnit5 удобна для группировки тестовых методов. Логически имеет смысл группировать вместе определённые типы тестов (типа InputIsXY, ErrorCases) или собрать в свою группу методы каждого теста (GetDesign и UpdateDesign).

**public** **class** **DesignControllerTest** {

@Nested

**class** **GetDesigns** {

@Test

**void** **allFieldsAreIncluded**() {}

@Test

**void** **limitParameter**() {}

@Test

**void** **filterParameter**() {}

}

@Nested

**class** **DeleteDesign** {

@Test

**void** **designIsRemovedFromDb**() {}

@Test

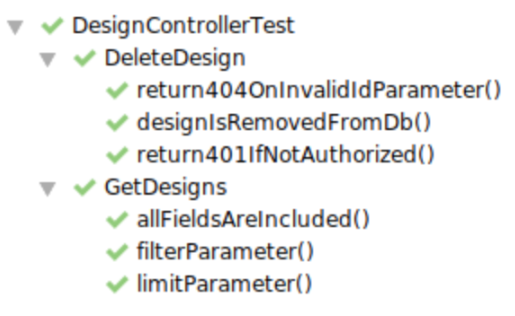
**void** **return404OnInvalidIdParameter**() {}

@Test

**void** **return401IfNotAuthorized**() {}

}

}



Читаемые названия тестов при помощи @DisplayName или обратных кавычек в Kotlin

В Java можно использовать аннотацию @DisplayName, чтобы дать тестам более читаемые названия.

**public** **class** **DisplayNameTest** {

@Test

@DisplayName("Design is removed from database")

**void** **designIsRemoved**() {}

@Test

@DisplayName("Return 404 in case of an invalid parameter")

**void** **return404**() {}

@Test

@DisplayName("Return 401 if the request is not authorized")

**void** **return401**() {}

}

В Kotlin можно использовать имена функций с пробелами внутри, если заключить их в обратные одиночные кавычки. Так вы получите читаемость результатов без избыточности кода.

@Test

**fun** `design **is** removed from db`() {}

Имитируйте внешние сервисы

Для тестирования HTTP-клиентов нам необходимо имитировать сервисы, к которым они обращаются. Я часто использую в этих целях [MockWebServer](https://github.com/square/okhttp/tree/master/mockwebserver) из OkHttp. Альтернативами могут служить [WireMock](http://wiremock.org/) или [Mockserver из Testcontainers](https://www.testcontainers.org/modules/mockserver/).

MockWebServer serviceMock = **new** MockWebServer();

serviceMock.start();

HttpUrl baseUrl = serviceMock.url("/v1/");

ProductClient client = **new** ProductClient(baseUrl.host(), baseUrl.port());

serviceMock.enqueue(**new** MockResponse()

.addHeader("Content-Type", "application/json")

.setBody("{\"name\": \"Smartphone\"}"));

ProductDTO productDTO = client.retrieveProduct("1");

assertThat(productDTO.getName()).isEqualTo("Smartphone");

Используйте Awaitility для тестирования асинхронного кода

[Awaitility](https://github.com/jayway/awaitility) — это библиотека для тестирования асинхронного кода. Вы можете указать, сколько раз надо повторять попытки проверки результата перед тем, как признать тест неудачным.

**private** **static** **final** ConditionFactory WAIT = await()

.atMost(Duration.ofSeconds(6))

.pollInterval(Duration.ofSeconds(1))

.pollDelay(Duration.ofSeconds(1));

@Test

**public** **void** **waitAndPoll**(){

triggerAsyncEvent();

WAIT.untilAsserted(() -> {

assertThat(findInDatabase(1).getState()).isEqualTo(State.SUCCESS);

});

}

Не надо резолвить DI-зависимости (Spring)

Инициализация DI-фреймворка занимает несколько секунд перед тем, как тесты могут стартовать. Это замедляет цикл обратной связи, особенно на начальном этапе разработки.  
  
Поэтому я стараюсь не использовать DI в интеграционных тестах, а создаю нужные объекты вручную и «провязываю» их между собой. Если вы используете внедрение в конструктор, то это самое простое. Как правило, в своих тестах вы проверяете бизнес-логику, а для этого DI не нужно.  
  
Более того, начиная с версии 2.2, Spring Boot поддерживает ленивую инициализацию бинов, что заметно ускоряет тесты, использующие DI.

Ваш код должен быть тестируемым

Не используйте статический доступ. Никогда

Статический доступ — это антипаттерн. Во-первых, он запутывает зависимости и побочные эффекты, делая весь код сложночитаемым и подверженным неочевидным ошибкам. Во-вторых, статический доступ мешает тестированию. Вы больше не можете заменять объекты, но в тестах вам нужно использовать моки или реальные объекты с другой конфигурацией (например, DAO-объект, указывающий на тестовую базу данных).  
  
Вместо статического доступа к коду положите его в нестатический метод, создайте экземпляр класса и передайте полученный объект в конструктор.

// Неправильно

**public** **class** **ProductController** {

**public** List<ProductDTO> **getProducts**() {

List<ProductEntity> products = ProductDAO.getProducts();

**return** mapToDTOs(products);

}

}

// Правильно

**public** **class** **ProductController** {

**private** ProductDAO dao;

**public** **ProductController**(ProductDAO dao) {

**this**.dao = dao;

}

**public** List<ProductDTO> **getProducts**() {

List<ProductEntity> products = dao.getProducts();

**return** mapToDTOs(products);

}

}

К счастью, DI-фреймворки типа Spring предоставляют инструменты, делающие статический доступ ненужным, автоматически создавая и связывая объекты без нашего участия.

Параметризуйте

Все релевантные части класса должны иметь возможность настройки со стороны теста. Такие настройки можно передавать в конструктор класса.  
  
Представьте, например, что ваш DAO имеет фиксированный лимит в 1000 объектов на запрос. Чтобы проверить этот лимит, вам надо будет перед тестом добавить в тестовую БД 1001 объект. Используя аргумент конструктора, вы можете сделать это значение настраиваемым: в продакшене оставить 1000, в тестировании сократить до 2. Таким образом, чтобы проверить работу лимита вам будет достаточно добавить в тестовую БД всего 3 записи.

Используйте внедрение в конструктор

Внедрение полей — зло, оно ведёт к плохой тестируемости кода. Вам необходимо инициализировать DI перед тестами или заниматься стрёмной магией рефлексий. Поэтому предпочтительно использовать внедрение через конструктор, чтобы легко контролировать зависимые объекты в процессе тестирования.  
  
На Java придётся написать немного лишнего кода:

// Правильно

**public** **class** **ProductController** {

**private** ProductDAO dao;

**private** TaxClient client;

**public** **ProductController**(ProductDAO dao, TaxClient client) {

**this**.dao = dao;

**this**.client = client;

}

}

В Kotlin тоже самое пишется намного лаконичнее:

// Правильно

**class** **ProductController**(

**private** **val** dao: ProductDAO,

**private** **val** client: TaxClient

){

}

Не используйте Instant.now() или new Date()

Не надо получать текущее время вызовами Instant.now() или new Date() в продакшн-коде, если вы хотите тестировать это поведение.

// Неправильно

**public** **class** **ProductDAO** {

**public** **void** **updateDateModified**(String productId) {

Instant now = Instant.now(); // !

Update update = Update()

.set("dateModified", now);

Query query = Query()

.addCriteria(where("\_id").eq(productId));

**return** mongoTemplate.updateOne(query, update, ProductEntity.class);

}

}

Проблема в том, что полученное время не может контролироваться со стороны теста. Вы не сможете сравнить полученный результат с конкретным значением, потому что он всё время разный. Вместо этого используйте класс Clock из Java.

// Правильно

**public** **class** **ProductDAO** {

**private** Clock clock;

**public** **ProductDAO**(Clock clock) {

**this**.clock = clock;

}

**public** **void** **updateProductState**(String productId, State state) {

Instant now = clock.instant();

// ...

}

}

В этом тесте вы можете создать мок-объект для Clock, передать его в ProductDAO и сконфигурировать мок-объект так, чтобы он возвращал одно и то же время. После вызовы updateProductState() мы сможем проверить, что в базу данных попало именно заданное нами значение.

Разделяйте асинхронное выполнение и собственно логику

Тестирование асинхронного кода — непростая штука. Библиотеки типа Awaitility оказывают большую помощь, но процесс всё равно запутан, и мы можем получить «моргающий» тест. Есть смысл разделять бизнес-логику (обычно синхронную) и асинхронный инфраструктурный код, если такая возможность имеется.  
  
Например, вынеся бизнес-логику в ProductController мы сможем запросто протестировать её синхронно. Вся асинхронная и параллельная логика останутся в ProductScheduler, который можно протестировать изолированно.

// Правильно

**public** **class** **ProductScheduler** {

**private** ProductController controller;

@Scheduled

**public** **void** **start**() {

CompletableFuture<String> usFuture = CompletableFuture.supplyAsync(() -> controller.doBusinessLogic(Locale.US));

CompletableFuture<String> germanyFuture = CompletableFuture.supplyAsync(() -> controller.doBusinessLogic(Locale.GERMANY));

String usResult = usFuture.get();

String germanyResult = germanyFuture.get(); } }